AD-A101 994 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 9/2
TRANSFORMATIONAL IMPLEMENTATION: AN EXAMPLE.(U)
MAY 81 R BALZER NSF-MCS76-8390
UNCLASSIFIED ISI/RR-79-79 NL

1 of 1
AD A
101994

END
DATE
FILMED
8-81
DTIC

ISI/RR-79-79
May 1981

Robert Balzer

Transformational Implementation: An Example

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA ISI 4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. ~~REPORT NUMBER~~ 14 ISI/RR-79-79	2. GOVT ACCESSION NO. AD-A101994	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 Transformational Implementation: An Example		5. ~~TYPE OF REPORT & PERIOD COVERED~~ 9 Research rept.
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Robert Balzer	15	8. CONTRACT OR GRANT NUMBER(s) NSF-MCS 768390
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS National Science Foundation 1800 G Street NW Washington, D.C. 20550	11	12. REPORT DATE May 1981
		13. NUMBER OF PAGES 60
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) -------	12 61	15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

----

18. SUPPLEMENTARY NOTES

----

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

documentation, implementation, program development, program manipulation, reliability, transformations.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

18

[illegible]

20. ABSTRACT

Program Manipulation Systems have been proposed as an alternative programming paradigm in which the PROCESSES of design and implementation are themselves the subject of study. These processes are captured and recorded to provide documentation of the program, the basis for its validation, and the framework within which future maintenance will occur. This extension of the conventional programming paradigm to include the development of the program as a computer processable object in addition to the program itself is quite profound. A key insight of the Program Manipulation approach is that transformations provide a sufficient basis for the design and implementation processes. Each development decision can be represented as a transformation applied to the program. Thus, a development is merely a linear sequence of transformations applied to the PROGRAM resulting from the previous stage of development. Each stage of the development corresponds to the transformation of a program treated as a specification into another treated as an implementation. Thus, development is an iterative (or sometimes a recursive) process of successive refinement in which a specification is gradually transformed into an implementation. This implies that the original specification of the program is itself a program (so that it can be transformed). To be a program, the specification language must have a formal semantics (thus precluding pseudo-code type languages) so that validity of transformations and of the development process is a meaningful concept. This report illustrates the use of an implemented Program Manipulation System to develop a highly declarative specification of a simple well known example into an efficient implementation and discusses the benefits and problems of this approach to program development.

ISI/RR-79-79

May 1981

Robert Balzer

Transformational Implementation: An Example

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA ISI 4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

This work was supported by National Science Foundation Grant MCS 768390. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

CONTENTS

INTRODUCTION

Much recent work has focused on the investigation of Program Manipulation Systems [1, 5, 6, 7, 8, 12, 14, 18] as an alternative programming paradigm in which the PROCESSES of design and implementation are themselves the subject of study. They are captured and recorded to provide documentation of the program, the basis for its validation, and the framework within which future maintenance will occur.

This extension of the conventional programming paradigm to include the development of the program as a computer processable object (in addition to the program) itself is quite profound. It is comparable to the early recognition that programs could themselves be treated as data, enabling computer languages to be developed. Correspondingly, by capturing and recording the development processes, a set of tools can be developed to use these processes as data.

Although such tools do not yet exist, it is easy to foresee some of their capabilities: automatically generated, up-to-date, and accurate documentation of the program relating the implementation back to its specification; explication of all the assumptions used within the development and identification of the decisions made therein; validation of an implementation based not on an analysis of the resulting program, but rather upon the process by which it was produced; maintenance performed by modifying the development process rather than by attempting to modify the optimized program; and automatic instrumentation to test the performance assumptions implicit in critical design and implementation decisions.

Before such possibilities can be realized, however, the development process that currently exists only within people's heads must be made explicit and recorded. How can this be accomplished?

A key insight of the Program Manipulation approach is that transformations provide a sufficient basis for the development process. Each development decision can be represented as a transformation applied to the program. Thus, a development is merely a linear sequence of transformations applied to the program. (Unfortunately, such linear sequences are unintelligible, and like programs, must be structured to be understandable.) But what programs are the transformations applied to? Since the object of the development is to produce a program, the resulting program is obviously not the one to which transformations are applied.

Instead, transformations are applied to the PROGRAM resulting from the previous stage of development. Each stage of the development corresponds to the transformation of a program treated as a specification into another treated as implementation. Thus, development is an iterative (and as we will see later, sometimes a recursive) process of successive refinement in which a specification is gradually transformed into an implementation.

This implies that the original specification of the program is itself a program (so that it can be transformed). To be a program, the specification language must have a formal semantics (thus precluding pseudo-code types of languages) so that validity of transformations and of the development process is a meaningful concept.

Since the motivation of this Program Manipulation approach is to capture and record the development process, it is essential that the specification be rather directly stated and that it be taken as the starting point for the development. Since the intent of a specification is to state WHAT is required, while the intent of an implementation is to state HOW those requirements are to be satisfied with minimal expenditure of computing resources, quite different languages for specification and implementation are implied.

This wide disparity between the specification and implementation language and the avoidance of determining HOW requirements should be satisfied within the specification suggest to us that the development (embodied as a sequence of transformations) must be humanly guided (rather than automatically generated), because such global optimization issues are not well-enough understood to automate (although this view is not universally held [4, 6, 17]). Thus, we have constructed an interactive system in which a user guides the system by specifying which transformations the system should apply. The rest of this report is a description of the development process in such an interactive Program Manipulation system (as embodied in a prototype we have built) through consideration of a simple example. This particular example was chosen to be simple enough to cover within this report, yet complex enough to demonstrate the type of issues that arise during development. In addition, only well-known examples were considered so that the one chosen did not have to be explained and motivated.

THE PROBLEM

Given a set of eight Queens, write a program that finds a way of positioning them on different squares of a chessboard so that no Queen may capture any other.

THE FORMAL SPECIFICATION

Before the development processes of design and implementation can begin, the problem must be expressed in a formal specification. This specification should express as much as possible WHAT the program is to do without expressing HOW it is to be accomplished. The WHAT specification will then be systematically converted into a HOW implementation during the development process.

We have developed a formal specification language [2] in which this problem can be directly stated. This language allows the definition of a world (Chess) in terms of the objects (the chess board, the squares of which it is composed, the rows and columns, the various chess pieces, etc.) of that world, the relationships that may exist among those objects (the immediate adjacency of two squares, the squares that comprise a row, pieces occupying a square, etc.), the actions that exist in that world (placing a piece on a square, moving a piece, capturing a piece, etc.), the constraints that the objects of the world must satisfy (two pieces can't occupy the same square), and the rules of inference within that world (a piece can capture another if it can move to the square occupied by that piece, etc.). These declarations define the environment within which the program will operate. An initial configuration of the objects in the world can be specified (in this case that the chess board is empty [no pieces are on the board] and that eight queens exist). The program portion of the specification then describes either the resulting configuration desired (preferred) or the behavior desired (acceptable). This latter option is provided because many real tasks cannot be simply stated in terms of a goal state, but rather are more naturally specified in terms of their desired behavior (such as a payroll system that periodically issues checks satisfying certain criteria). These behavioral specifications would naturally contain as much "resulting configuration" description as possible so as to least constrain the ultimate implementation. As the development proceeds, the "resulting configuration" portions are converted into behavior specifications, which are then specialized and optimized.

For the Eight Queens problem, the formal specification is shown in Figure 1. For the sake of conciseness and perspicuity the definition of objects, relationships, and actions has been suppressed, as has the specification of the initial configuration (which defines the structure of the chess board, the fact that no pieces are on any of the squares, and the existence of eight queens) and the inference rule defining Queen-Capture.

```
PROGRAMS:
        QUEENS: [LAMBDA (QUEEN-SET)
                         (LOCAL (BOARD-POSITION)
                                (FOR QUEEN IN-SET QUEEN-SET DO
                                     (DETERMINE BOARD-POSITION FROM
                                                (CHESS-BOARD BOARD-POSITION))
                                     (ASSERT (PIECE-ON-BOARD QUEEN
                                                             BOARD-POSITION]

CONSTRAINT: TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE
            PATTERN: (AND (PIECE-ON-BOARD PIECE#1 BOARD-POSITION)
                          (PIECE-ON-BOARD PIECE#2 BOARD-POSITION))
            PATTERN-VARIABLES: (PIECE#1 PIECE#2 BOARD-POSITION)

CONSTRAINT: QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN
            PATTERN: (AND (PIECE-ON-BOARD QUEEN#1 BOARD-POSITION#1)
                          (PIECE-ON-BOARD QUEEN#2 BOARD-POSITION#2)
                          (QUEEN-CAPTURE BOARD-POSTION#1
                                         BOARD-POSTION#2))
            PATTERN-VARIABLES: (QUEEN#1 QUEEN#2 BOARD-POSITION#1
                                BOARD-POSITION#2)
```

Figure 1

The formal specifications indicate that subject to two constraints (that two pieces can't occupy the same square and that queens cannot be placed so that they can capture each other), each queen in the set of (presumably eight) queens is to be placed somewhere on the chess board. Although this specification is not the most abstract possible (a completely "resulting state" specification is quite straightforward), and although a minor implementation restriction has been imposed (the requirement of queens not capturing each other is only logically imposed on the resulting solution and need not necessarily be true while the solution is being constructed, as is the case in the formal specification of Figure 1), this specification has been chosen to reduce the amount of unconventional specification constructs considered in this example.

THE DEVELOPMENT PLAN

The development of this specification into an implementation progressed in three main phases: explication, reorganization, and representation selection. In the explication phase, implicit structures within the specifications are made explicit and constraints are dealt with as early as possible in an attempt to gain an understanding of the algorithmic structure implied by the specification. In the reorganization phase, the sources of computational expense are identified and the program reorganized to mitigate these expenses. Representations suitable for the reorganized programs are selected in the third phase.

This phase-based conceptualization of the development process is not yet part of our prototype system and is introduced here tc help the reader understand our development plans. We theorize that in more complex tasks many cycles of this basic plan occur (it is also clear that the representation selection may precede the reorganization). If so, then some structure must exist among these cycles. Such structure, arising partially a priori (plan) and partially a posteriori (documentation), represents the explanation of the development.

Currently, a much more primitive development explanation is maintained by the system (see Appendix A). It consists of a linear sequence of state descriptions. Each description is composed of the state name, a comment entered by the developer, and the action taken in that state (such as applying a transformation or loading the initial specification). Structure is added to the linear sequence only when the action taken within the state fails. Development proceeds within the suspended state until the failing action succeeds.

For the development explanation shown in Appendix A, the explication phase corresponds to states 1 through 7, the reorganization phase corresponds to states 8 and 9, and the representation phase corresponds to state 10.

THE DEVELOPMENT PROCESS

The development of the implementation from the formal specification is described informally here. The actual form of the program at each step is given in Appendix B, which is organized as pairs of program displays that highlight (in bold face) the changes from state N into state N+1. These pairs of program displays are produced by the system as part of the automatic documentation of a development [9].

In the text that follows, the informal description of the transition into a state will be preceded by its state description as given in Appendix A.

[State-2 (Unfold both constraints)
(Apply transformation: Unfold-Constraint)]

As the first step in making implicit structures explicit, both constraints contained in the original formal specifications are "unfolded." That is, rather than relying on the interpreter of the language to check the constraints after each (relevant) operation and to backtrack in case the constraint is violated, an analysis is performed to determine where explicit checks *should be inserted in the program* (after assertions that could affect the truthfulness of the constraint predicate), and the appropriate checking and backtracking code is added there. This analysis is performed by the unfold-constraint transformation. As part of its analysis, a simplification of the constraint predicate is performed (because the assertion the check follows will be true and need not be rechecked). If this predicate is satisfied, then the constraint has been violated and the call to constraint-violation is executed, which invokes the backtracking mechanism to reevaluate the most recent nondeterministic statement (the choice of a board position on which to place the Queen).

Both constraints are unfolded by this transformation, and since each contains two instances of the fact being asserted, each generates two checks which are inserted in the program. The constraints, having been unfolded, are removed from the program.

[State-3 (Simplify: remove redundant unfolded constraint checks)
(Manual effort)]

Because of the symmetry that existed in the constraint patterns, one of the two checks generated by the previous transformation for each constraint is redundant. The current system does not include an automatic simplifier, so either transformations for this particular type of simplification must be applied or else the redundant code must be manually removed. The latter option was chosen to illustrate this facility within the system.

It is assumed that situations will inevitably arise for which the appropriate transformation does not already exist within the catalog. Therefore, the developer may either define a new transformation (thus extending the catalog) or modify the program directly through an interactive editor (i.e., manually modify the program). It must be recognized that both options result in an unvalidated modification of the program (merely defining a transformation does not ensure its validity). In both cases the unvalidated step becomes part of the documentation of the development, which can later be reviewed by the others and judged acceptable or not.

The redundancy of the first pair is based on simple renaming of free variables, while the second also depends upon determining that Queen-Capture is a symmetric relation. The second element of each pair of checks was manually edited out of the program.

[State-4 (Make backtracking explicit so that it can be minimized)
(Apply transformation: Unfold-Consequential-Backtracking)]

The second step in making implicit structure explicit is now attempted by applying the transformation to unfold backtracking (other backtracking transformations can be found in references [10, 13]). This transformation converts the implicit control structure necessary to support resumption of control at a nondeterminism point from an arbitrary failure into a format in which the nondeterminism is embedded in an iterative loop through all the possibilities searching for an acceptable one as determined by the loop body containing all the possible failure points reexpressed as loop continuation statements.

The activation pattern for this transformation assumes a recursive format for the routine containing the nondeterminism. Unfortunately, when the transformation is applied to the program in State-3, this activation pattern fails to match. This causes the system to ask the developer whether he would like to modify ("jitter") the program so that the activation pattern will match or abort the application of the current transformation. The developer responded that jittering was desired. The system then enters a subgoaling mode in which further development proceeds under the direction of the developer until the suspended activation pattern successfully matches the modified program. At that point the development pops out of the subgoal mode and continues application of the suspended transformation.

[State-4-1 (Convert iteration to recursion)
(Jitter transformation: Make-set-iteration-recursive)]

In the subgoal development the developer applies a transformation (recorded as a jitter transformation because the developer is attempting to get the program to conform to the requirements of a suspended transformation) that converts the iteration to a recursion.

It is instructive to digress for a moment and consider in detail the application of this transformation, which is shown in Figure 2. The transformation contains a comment, an activation pattern, a list of modifications and declaration of variables used within the transformation. In addition, it could contain properties that the program and/or data had to satisfy in order for the transformation to be applicable or properties known to be true after the transformation was applied.

The pattern contains variables and literals (all names not declared to be variables). The variables will be matched against a single expression in the program to which the transformation is applied or against a sequence of expressions (if the variable begins with an exclamation mark). When this pattern is applied to the program in State-3, a unique match is found in which SET1 is bound to QUEEN-SET, P2 is bound to (BOARD-POSITION), the segment variable !S1 is bound to everything following the DO in the FOR statement (the DETERMINE and inner LOCAL statements), etc. If more than a single match were found, the developer would have been asked which match to use. If no match were found, the developer would have been asked whether the program should be jittered or the transformation aborted.

```
                    MAKE-SET-ITERATION-RECURSIVE

COMMENT: CONVERT SIMPLE SET ITERATION THROUGH THE (ONLY)
         PARAMETER INTO A RECURSION

PATTERN: [LAMBDA (SET 1)
                 (LOCAL P2 (FOR O1 IN-SET SET1 DO !S1)
                       !S2]

MODIFICATIONS: [(BIND R1 FROM FNAME)
               (BIND P3 FROM (CONS O1 P2))
               (REPLACE-PATTERN
                ((LAMBDA (SET1)
                         (LOCAL P3 (TERMINATION-TEST:
                                     (IF (EMPTY SET1)
                                         THEN
                                         !S2
                                         (RETURN)))
                                   (REMOVE O1 FROM SET 1)
                                   !S1
                                   (RECURSIVE-CALL: (R1 SET1]

PATTERN-VARIABLES: (SET1 !S1 O1 P2 !S2 P3 R1)
```

Figure 2

Following the pattern match, the applicability properties to be satisfied are checked (there are none in this transformation). These properties fall into two categories--properties that must be satisfied before the transformation can be applied and properties that eventually must be satisfied to validate the applicability of this transformation but need not be considered immediately. Such properties are quite important because they build up "requirements" on the program and/or data that must eventually be satisfied, but that because they can be delayed, can be used as guidance for the subsequent development. If any immediate properties were not satisfied, then the system would attempt to prove them (currently only through special-purpose property provers). Failing that, it would enter a subgoaling mode until further development established the immediate property.

After the applicability properties are satisfied (or delayed), the modifications are performed. These modifications are a linear sequence of actions. The prototypical action is to replace the portion of the program matched by the applicability pattern with some new pattern composed of literals and variables. The variables used in this replacement pattern can be either bound by the applicability pattern (such as SET1) or ones calculated from those variables (such as R1 and P3) through the BIND action. This sequence of BIND actions calculating new values that become part of the replacement pattern is quite typical of the transformations we have studied (and separates them from simpler so called "syntactic" transformations involving only pattern replacement).

In addition to these actions, other transformations can be applied (providing a means to package transformations), arbitrary functions invoked, or a simple plan established through goals to be achieved.

In the current transformation, the BIND action is used to calculate values for the recursive call of the program being transformed (in variable R1) and for the addition of the program iteration variable (QUEEN) to the declaration of variables in the LOCAL statement (in the transformation variable P3). These values are then used, along with several found directly by the applicability pattern, to form the replacement pattern.

The resulting program has annotations in the form of labels (names ending with a colon), which describe the teleological function of that portion of the program [16]. These annotations may well help later transformations access and analyze appropriate parts of the program. But as these annotations are currently part of the program text, they must be dealt with by succeeding transformations whether they are interested in those annotations or not. This has proved most bothersome. One solution used in the MENTOR system [11] is to place these annotations into orthogonal dimensions accessible only by special commands so that the annotations are invisible for those uninterested. We expect to employ this solution for all annotations including the maintenance of properties.

After applying the jittering transformation of State 4-1, the modified program successfully matches the applicability pattern of the suspended unfold-consequential-backtracking transformation, and so processing of this transformation is resumed. The transformation (shown in Figure 3) is similar in structure to the previous transformation as its modifications consist of a series of BIND actions followed by a REPLACE-PATTERN action. Unlike the previous transformation, this one contains some NECESSARY-PROPERTIES that must be satisfied before the transformation can be applied. The properties are arbitrary predicates applied to the various objects of the system being implemented (such as its programs, code segments, and data structures). In this transformation there are four instances of the same property applied to single (but different) arguments consisting of a code segment identified by the match of the transformation-applicability pattern. Aside from a small number of built-in properties directly relating to the semantics of the specification language (and produced and maintained by an analysis package soon to be incorporated within the system), all other properties must either be defined in terms of other properties (theorem-proving techniques will be used to determine whether or not the property is satisfied) or self-defined through direct generation as the result (the KNOWN property) of some transformation(s) and/or deduction through special-purpose property provers. Typically for self-defined properties, no explicit formal definition exists.

UNFOLD-CONSEQUENTIAL-BACKTRACKING

```
COMMENT: MAKE BACKTRACKING EXPLICIT

PATTERN: (LOCAL V1 (TERMINATION-TEST: (IF P2 THEN !S5 (RETURN)))
                 !S1
                 (DETERMINE O1 FROM P1)
                 !S2
                 (RECURSIVE-CALL: S3)
                 !S4)

NECESSARY PROPERTIES: ((CONSEQUENTIAL-NON-DETERMINISM-FREE !S1)
                       (CONSEQUENTIAL-NON-DETERMINISM-FREE !S2)
                       (CONSEQUENTIAL-NON-DETERMINISM-FREE !S4)
                       (CONSEQUENTIAL-NON-DETERMINISM-FREE !S5))

MODIFICATIONS:[(BIND !S2* FROM
                       (UNFOLD-CONSEQUENTIAL-BACKTRACKING-BUILDER !S2))
                 [BIND !S2UNDO FROM (UNDO-OF (ACTIVE-PREDECESSORS-OF
                                              NIL !S2]
                 [BIND !S1UNDO FROM (UNDO-OF (ACTIVE-PREDECESSORS-OF
                                              NIL !S1]
                 (BIND !S5* FROM (UNFOLD-CONSEQUENTIAL-BACKTRACKING-BUILDER
                       !S5 T))
                 (REPLACE-PATTERN
                 ((LOCAL V1 (TERMINATION-TEST: (IF P2 THEN !S5*
                                                    (EXIT SUCCESSFUL)))
                       !S1
                       (FOR ALL P1 THEREIS !S2*
                            (IF (SUCCESSFUL (RECURSIVE-CALL: S3))
                                THEN
                                (LOOP-RETURN CHOICE-ACCEPTED)
                                ELSE
                                (UNDO-ACTIONS: !S2UNDO)
                                (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                            THEN
                            !S4
                            (EXIT SUCCESSFUL)
                            ELSE
                            (UNDO-ACTIONS: !S1UNDO)
                            (EXIT UNSUCCESSFUL]

PATTERN-VARIABLES: (!S1 !S2 !S4 !S2* !S2UNDO !S1UNDO V1 P2 P1
                       S3 O1 !S5* !S5)
```

Figure 3

The property (CONSEQUENTIAL-NON-DETERMINISM-FREE) used in this transformation is defined only through its special-purpose property prover. Informally, the property means that the code segment to which it is applied does not contain any "meaningful" nondeterminism. This is important because this transformation rearranges the program so that any "backtracking point" is part of the still active loop; backtracking can then be accomplished by merely continuing that loop (after undoing any actions taken within the loop). It assumes that only one such backtracking point exists, and uses the property to validate this assumption. The concern is that only a single backtracking point exists and the existence of "incidental" nondeterminism is of no consequence. In the program of State 4-1 there are two nondeterministic statements, the selection of a queen and the determination of where to place it on the chessboard. The first of these is incidental in that it doesn't matter which queen is selected. But the second is consequential because the course of subsequent processing is highly dependent upon the choice made. Thus, in backtracking, only the nondeterminism of the selection of a board position should be considered as the other, incidental choice of a queen doesn't affect the subsequent processing.

After the applicability pattern of the transformation has been matched, the system attempts to verify that the necessary properties are satisfied. The first property is CONSEQUENTIAL-NON-DETERMINISM-FREE applied to the segment !S1 which is bound to the statement (REMOVE QUEEN FROM QUEEN-SET). The method of verifying this property is to invoke its special-purpose property prover, which fails because it is unable to tell that this nondeterminism is incidental. Because a necessary property couldn't be verified, the system enters a subgoal mode (through a self-generated ACHIEVE command resulting in State 4-2) under which further development will continue until the necessary property is achieved. An appropriate message to this effect is given to the developer.

[State 4-2-1 (Mark "REMOVE" as incidental nondeterminism)
(Apply transformations: Mark-incidental-non-determinism)]

The developer responds to this problem by applying a transformation that marks the REMOVE statement as incidental (by converting it to REMOVE*). This transformation has as a required property that the choice of the object being removed (Queen) is incidental. Since this is a required rather than an immediate property, it need not be verified immediately, and so it is added as an UNPROVED-PROPERTY of the current state. Such properties must be either proved (by one of the methods described previously) or claimed (assumed by the user to be true) before the development is completed. Any claims (developer assumptions) become part of the documentation of the development.

After each step of the development in the subgoal mode, the system attempts to determine whether the property to be achieved can be verified. Here, the appropriate method is to reinvoke the special-purpose property prover on code segment !S1. However, the transformation just applied modified this code segment and the new value must be used. This is accomplished by rematching the suspended applicability pattern to obtain the appropriate code segment. In fact, since the subgoal development logically procedes the suspended transformation, the suspended transformation is reprocessed after each subgoal development step.

With !S1 rebound to (REMOVE* QUEEN FROM QUEEN-SET), the special-purpose property prover is successful in verifying the CONSEQUENTIAL-NON-DETERMINISM-FREE property.

Verification of the first instance of the CONSEQUENTIAL-NON-DETERMINISM-FREE property completes the processing of the subgoal ACHIEVE state, and processing of State 4 resumes with attempts to verify the other instances of this property, all of which succeed.

Processing then continues with the modification steps. The second and third BIND actions find the active statements within some code segment and then build a sequence that undoes their effects (so that when control is returned to the "backtracking point" [now in the form of an interactive loop] the program state has been restored to its state when control was last there). The first and last BIND actions locate any CONSTRAINT-VIOLATION statements occurring in the program and replace them by the undo of the active statements preceding the CONSTRAINT-VIOLATION statement (and following the nondeterministic choice) followed by a loop continuation statement (to the loop being introduced by this transformation). After all the BIND actions have been processed, the replacement pattern is constructed and substituted for the portion of the program matched by the transformation's applicability pattern.

[State 5 (Assimilate constraint into generator)
(Apply transformation: Assimilate-test-in-thesis-loop)]

The final stage in explicating the underlying structure of the algorithm is to incorporate into its selection the constraints that a board position must satisfy. Here, these constraints are tested after a queen is placed on the board at the selected position (because the constraints were originally stated in terms of pieces on the board), and if the position is unacceptable, then the queen is removed from the board and another selection made. In general, more efficient processing results when a selection is based on all such restrictions that it must satisfy. Toward this end, the developer attempts to apply a transformation to assimilate one of these restrictions into the generator of board positions. (This same transformation will be applied again to assimilate the other restrictions.) Unfortunately the applicability pattern of this transformation requires that the restriction being assimilated be the first statement of the body of the for-loop in which the generator occurs, and this fails to match the program. As before, the system asks the developer whether jittering is desired, and when affirmed, initiates a subgoal development (which will take several steps before the suspended applicability pattern can be matched).

[State 5-1 (Merge the locals)
(Jitter transformation: Merge-locals)]

As the first step in the jittering process, the developer attempts to merge the LOCAL inside the for-loop with the outer LOCAL. Again the applicability pattern fails because the two LOCALs being merged must be separated by only a single level of nesting (here the inner LOCAL is inside the for-loop, which is inside the LOCAL), and so (after confirmation by the developer) another level of jittering is initiated.

**[State 5-1-1 (Extract LOCAL from for-loop)
(Jitter transformation: Extract-local-from-for-loop)]**

The developer applies a transformation that extracts the LOCAL from within the for-loop. This transformation enables the suspended applicability pattern of the Merge-locals transformation to succeed and so ends this level of jittering.

The suspended applicability pattern matched the jittered program and the transformation's modification steps are performed. First a BIND statement is used to calculate the combined set of variable declarations, and then the replacement pattern is formed which contains the combined declarations and in which the statements of the inner LOCAL are embedded within the outer one. (As noted previously, the current system does not yet employ orthogonal annotation dimensions [11]. Here the LOCAL statements are used solely to scope variables. If this variable scoping were handled as an annotation it would not impede the development as it does here.)

**[State 5-3 (Move constraint test ahead of assertion)
(Jitter transformation: Move-constraint-uphill)]**

The restriction has now been moved to the top level of the for-loop body, but it follows the assertion (rather than preceding it, as required by the suspended assimilation applicability pattern). The developer applies the Move-constraint-uphill to interchange the order of the restriction and the assertion preceding.

To interchange these two statements, their mutual interactions must be revised to select the new ordering. This leaves the assertion unchanged since it has no dependence on the restriction. However, both the predicate and body of the restriction must (in general) be updated to reflect knowledge of the existence of the assertion that now follows them. Thus, if the predicate depended, in part, on the existence of the assertion, it must be modified to incorporate this dependence without actually accessing the data (because it will not yet exist). In the current case, the predicate is independent of the assertion, and so is not changed. On the other hand, the THEN clause denies the assertion before forcing another iteration of the selection loop, and thus, is highly dependent upon the assertion. The semantics of the program is maintained if the denial is removed (whenever another iteration is forced, the assertion will not exist). These modifications to the program are calculated by special-purpose analysis routines (called as part of the transformation's BIND actions), incorporated into the replacement pattern, and substituted into the program.

**[State 5-5 (Simplify: suppress null else clause)
(Jitter transformation: Else-suppression)]**

Although the restriction is now in the right location (as the first statement in the for-loop body), the suspended applicability pattern still doesn't match because it requires the restriction to be an IF-THEN statement (with no ELSE clause), and this restriction has an ELSE clause whose body is NIL.

The fact that this match fails because of this trivial problem points up two difficulties with the current system. First, no automatic simplification exists. Such simplification would (if it existed) certainly have removed this null ELSE clause when it was created (by the *unfold-constraint transformation* in State 2). Second, given that a simple mismatch exists between the applicability pattern and program, rather than have jittering be a manual process (as it currently is), the system should automatically jitter the program so that the desired transformation can be applied (we are working to resolve both these deficiencies). Here, the developer had to apply a simplification transformation to remove the null ELSE clause. This modification enabled the suspended applicability pattern to succeed, and so, processing of the jittering subgoal development is completed.

The suspended applicability pattern succeeds, and the replacement pattern is formed by composing a new loop predicate consisting of the conjunctions of the old loop predicate and the negation of the restriction predicate. The rest of the restriction (the IF-THEN structure and the body of the THEN clause) is deleted and its semantics are now part of the loop itself.

[State 6 (Assimilate remaining constraint into loop generator)
(Apply transformation: Assimilate-test-in-thesis-loop)]

The sequence of State 5 (without having to bother with the LOCALs) is repeated (interchanging the order of the restriction and the assertion, and suppressing the null ELSE clause) to incorporate the remaining restriction into the loop generator.

[State 7 (Simplify: remove embedded AND in loop generation)
(Apply transformation: Simplify-AND)]

The embedded conjunction within the loop predicate is merged with the outer conjunction. Automatic simplification would, when added to the system, remove the need for this step.

[State 8 (Maintain acceptable board positions incrementally)
(Apply transformation: Calculate-predicate-incrementally)]

The simplification performed in the previous state completes the explication phase of the development whose purpose was to reveal the underlying structure of the algorithm implicit in the original specification. This structure is quite clear in State 7. A simple recursive program exists in which on each recursive level, a queen is removed from the set of queens and placed on a position on the chess board not already occupied and not capturable by any queen already on the chess board. The recursion is then carried out at the next level and if it (and all its recursive calls) is successful, the algorithm terminates. If not, the queen is removed from the board and another position selected.

The computationally expensive part of this algorithm is finding an acceptable board position. The developer recognizes that this same calculation (in a slightly altered environment with an extra queen placed on the board) is carried out at each level, and decides that rather than repeat these similar calculations, the set of acceptable board positions should be maintained incrementally. That is, as actions affecting the membership of the set occur within the program, appropriate maintenance actions are inserted to update the set membership accordingly. The actions that can affect set membership are the assertion and/or denial of facts which interact with the set definition predicate. Such statements are either preceded (assertions) or followed (denials) by the necessary maintenance actions. Furthermore, the iteration through the elements of the set is changed from a generative format (FOR ALL <predicate>...) to simple membership in a preexisting set (FOR <X> IN-SET <set>...).

These modifications are all made automatically by the Calculate-predicate-incrementally transformation. The calculation of the precise predicate to use to update the set membership is quite complex (see [3] for the details and [15] for the foundations of these ideas), but the general idea is straightforward: given the newfound truth (or falseness) of a fact, what else must necessarily be true to make the set definition pattern change its value (that is, become true or become false, or equivalently, to have the support of the corresponding set element(s) change) and to ensure that no other support exists for these elements? That is, what must be true to detect the creation of the first support of an element or the deletion of its last support? Only then should it be added to, or deleted from, the set.

For the set of board positions, this maintenance predicate, following the placement of a queen on the board or its removal, is determined to be the set of board positions that are on the chess board, are not occupied by any other piece, ARE capturable by the queen being placed on the board or removed from it, and are not capturable by any other queen already on the board.

Actually, since the PIECE-ON-BOARD pattern occurs twice in the set definition predicate, two maintenance action loops are generated. The first of these is concerned only with the position occupied by the queen (or about to be occupied by it), while the second maintenance loop is the one described above.

[State 9 (Simplify: remove redundant loop from maintenance actions)
(Manual effort)]

The single position handled by the first maintenance loop is also handled by the second, more general, loop. It is therefore redundant, and the developer used the editor to manually remove the first maintenance loop (both the occurrence preceding the assertion and the occurrence following the denial).

[State 10 (Pick a representation for board positions)
(Manual effort)]

State 9 completes the reorganization phase of the development accomplished by deciding to employ incremental set maintenance. The final phase deals with determining representations appropriate for the processing. At this stage, the expensive part of the processing is determining within the maintenance actions whether particular positions are capturable by any queen. Is there a representation in which this operation can be more easily performed?

The time has come for the developer to introduce a little creative magic (it is at just these points that a purely automatic approach seems most suspect). By switching the viewpoint from positions to lines (i.e., rows, columns, and diagonals) the problem is greatly simplified. That is, rather than incrementally maintaining the set of remaining board positions explicitly, the set of remaining lines is incrementally maintained and the remaining board positions are generated from them (as the intersection of four of the remaining lines: a row, a column, and two diagonals). In this representation, maintaining the set of remaining lines when a queen is placed on the board (or removed from it) merely involves deleting (or adding) the corresponding lines (the row, the column, and two diagonals) without any search. Furthermore, the iteration through the remaining board positions becomes a quadruply nested loop through the remaining rows, columns, and left and right diagonals finding four lines that intersect at a single position. Since a position is uniquely determined by a row and column, the inner two loops can be replaced by checks of whether the corresponding diagonals remain.

These modifications produce the program in State 10. It was produced by manual editing, but we are investigating how this step can be formulated as a representation alteration transformation (called type transformations [19]).

REMAINING OPTIMIZATIONS

A few steps remain to complete the optimization and convert the program into conventional form. They include removing the outer loop through the rows (since each row must have a queen and failure to place a queen in a row cannot be resolved by reconsidering that row again later), eliminating the use of the set of queens used only to determine termination (by using any one of the other sets which become empty at the same time), explicitly collecting the set of queen placements as the result of the computation rather than having it be implicit in the set of assertions in the data base, and using lists (or arrays) instead of sets.

We have not yet worked on these optimizations pending resolution of the representation selection issue.

CONCLUSIONS

Using a transformation system to develop the implementation of a small, but nontrivial example, such as the Eight Queens problem presented here, is both instructive and disconcerting.

Developing an implementation through the application of formalized transformations forces a more careful consideration of the strategy to be employed, and the tradeoffs involved. This highly beneficial result represents a shift in focus away from maintaining consistency, which almost completely consumes today's programmers, toward a concern with tradeoffs between alternative implementations. With the system assuming the responsibility for maintaining consistency, the developer should be free to concentrate on these higher level issues. Such consideration of the implementation tradeoffs heightens the need for adequate specifications that merely define the required behavior without determining how it is to be achieved.

However, it is quite evident from the development presented here that the developer has not been freed to consider implementation tradeoffs. Instead of a concern for maintaining consistency, the equally consuming task of directing the low-level development has been imposed. While the correctness of the program is no longer an issue, keeping track of both where one is in a development and how to accomplish each step in all its fine detail diverts attention from the tradeoff question.

It is quite clear that if transformation systems are to become useful, this difficulty must be removed. Automatic simplification and jittering, as discussed in this report, will help considerably (see also [12]). But equally important is the ability to state, represent, refine, and display implemention plans. The current lack of an adequate framework in which the development proceeds is a major source of conceptual overload.

Major improvements are also needed in the documentation of developments to make them understandable. The ability to highlight changes between successive states, as illustrated in Appendix B, is but a first step. The structure of the development plan must become an integral part of its documentation and understanding.

Finally, if implementation tradeoffs are to gain prominence, and if systems developed via transformations are to be maintained, the ability must be created to replay a slightly altered development (i.e., the altered development becomes the implementation plan to be carried out largely or completely automatically).

All of the above benefits and problems are present only on the assumption that people are involved in the development process. If this process were totally automated, then none of these issues would arise. However, given the growing concern with starting a development from a very high-level (and largely noncomputational) specification, and the paucity of information known about strategic optimization that is central to implementing such specifications, it seems unlikely that fully automatic systems could deal effectively with such specification languages. Rather, one would expect to see a gradual raising of the level of the languages that can be automatically optimized. Thus, the role of interactive transformation systems will be to provide a validated mapping between the high-level specifications and the "programming language" from which automatic optimization can proceed.

APPENDIX A
Development Documentation

```
(STATE-1   (ENTER SPECIFICATION FOR EIGHT QUEENS PROBLEM)
           (LOAD FILE EIGHT-QUEENS.SPEC))

(STATE-2   (UNFOLD BOTH CONSTRAINTS)
           (APPLY TRANSFORMATION . UNFOLD-CONSTRAINT))

(STATE-3   (SIMPLIFY: REMOVE REDUNDANT UNFOLDED CONSTRAINT CHECKS)
           (MANUAL EFFORT))

(STATE-4   (MAKE BACKTRACKING EXPLICIT SO THAT IT CAN BE MINIMIZED)
           (APPLY TRANSFORMATION . UNFOLD-CONSEQUENTIAL-BACKTRACKING))

      (STATE-4-1 (CONVERT ITERATION TO RECURSION)
                 (JITTER TRANSFOMATION . MAKE-SET-ITERATION-RECURSIVE))

      (STATE-4-2 (ACHIEVE A NECESSARY PROPERTY OF A TRANSFORMATION)
                 (ACHIEVE THE FOLLOWING PROPERTY:
                          CONSEQUENTIAL-NON-DETERMINISM-FREE
                          REMOVE QUEEN FROM QUEEN-SET)))

           (STATE-4-2-1 (MARK "REMOVE" AS INCIDENTAL NON-DETERMINISM)
                        (APPLY TRANSFORMATION .
                               MARK-INCIDENTAL-NON-DETERMINISM))

           (STATE-4-2-2
             NIL
             ((CONSEQUENTIAL-NON-DETERMINISM-FREE (REMOVE* QUEEN FROM
                                                           QUEEN-SET))
              PROVED BY NO-CONSEQUENTIAL-NON-DETERMINISM))

      (STATE-4-3
        (ACHIEVE A NECESSARY PROPERTY OF A TRANSFORMATION)
        ((CONSEQUENTIAL-NON-DETERMINISM-FREE
           (LOCAL . . . )
         PROVED BY NO-CONSEQUENTIAL-NON-DETERMINISM))

      (STATE-4-4 (ACHIEVE A NECESSARY PROPERTY OF A TRANSFORMATION)
                 ((CONSEQUENTIAL-NON-DETERMINISM-FREE)
                 PROVED BY NO-CONSEQUENTIAL-NON-DETERMINISM))
```

```
(STATE-5 (ASSIMILATE CONSTRAINT INTO GENERATOR)
         (APPLY TRANSFORMATION . ASSIMILATION-TEST-IN-THEREIS-LOOP))

     (STATE-5-1 (MERGE THE LOCALS)
                (JITTER TRANSFORMATION . MERGE-LOCALS))

          (STATE-5-1-1 (EXTRACT LOCAL FROM FOR LOOP)
                       (JITTER TRANSFORMATION .
                               EXTRACT-LOCAL-FROM-FOR-LOOP))

     (STATE-5-3 (MOVE CONSTRAINT TEST AHEAD OF ASSERTION)
                (JITTER TRANSFORMATION . MOVE-CONSTRAINT-UPHILL))

     (STATE-5-5 (SIMPLIFY: SUPPRESS NULL ELSE CLAUSE)
                (JITTER TRANSFORMATION . ELSE-SUPPRESSION))

(STATE-6 (ASSIMILATE REMAINING CONSTRAINT INTO LOOP GENERATOR)
         (APPLY TRANSFORMATION . ASSIMILATE-TEST-IN-THEREIS-LOOP))

     (STATE-6-1 (MOVE REMAINING CONSTRAINT AHEAD OF ASSERTION)
                (JITTER TRANSFORMATION . MOVE-CONSTRAINT-UPHILL))

     (STATE-6-3 (SIMPLIFY: SUPPRESS NULL ELSE CLAUSE)
                (JITTER TRANSFORMATION . ELSE-SUPPRESSION))

(STATE-7 (SIMPLIFY: REMOVE EMBEDDED APAND IN LOOP GENERATOR)
         (APPLY TRANSFORMATION . SIMPLIFY-APAND))

(STATE-8 (MAINTAIN ACCEPTABLE BOARD POSITIONS INCREMENTALLY)
         (APPLY TRANSFORMATION . CALCULATE-PREDICATE-INCREMENTALLY))

(STATE-9 (SIMPLIFY: REMOVE REDUNDANT LOOP FROM MAINTENANCE ACTIONS)
         (MANUAL EFFORT))

(STATE-10 (PICK <ROW# COLUMN#> AS REPRESENTATION OF BOARD POSITION.
                RECOGNIZE THAT COMPONENTS ARE ORTHOGONAL AND CAN BE
                INDEPENDENTLY SELECTED.  FURTHERMORE, RECOGNIZE THAT
                ROW AND COLUMN DETERMINE NE-DIAGONAL AND SE-DIAGONAL.
                FINALLY, RECOGNIZE THAT POSSIBLE BOARD-POSITIONS CAN BE
                GENERATED FROM THE INTERSECTION OF THE REMAINING ROWS,
                COLUMNS, AND DIAGONALS, AND THAT INCREMENTAL UPDATE
                OF POSSIBLE-BOARD-POSITIONS MERELY INVOLVES REMOVING
                THE CHOSEN ROW, COLUMN, AND TWO DIAGONALS BECAUSE
                THE MAPPING IS ONE-TO-ONE IN BOTH DIRECTIONS.)
          (MANUAL EFFORT))
```

APPENDIX B
Highlighted State Transitions

This appendix contains pairs of program displays in which the changes from one state into another are highlighted in boldface type. The deleted and/or modified text is highlighted to show the changes out of a state, while the additions and modifications are highlighted to show the changes into the next state. These pairs of program displays are placed on facing pages for ease of comparison. They were produced automatically as part of the documentation of the development [9].

CHANGES OUT OF STATE-1

```
(LAMBDA (QUEEN-SET)
      (LOCAL (BOARD-POSITION)
           (FOR QUEEN IN-SET QUEEN-SET DO
               (DETERMINE BOARD-POSITION FROM
                       (CHESS-BOARD BOARD-POSITION))
               (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION)
```

CHANGES INTO STATE-2

```
(LAMBDA (QUEEN-SET)
      (LOCAL (BOARD-POSITION)
            (FOR QUEEN IN-SET QUEEN-SET DO
                (DETERMINE BOARD-POSITION FROM
                         (CHESS-BOARD BOARD-POSITION))
                (LOCAL (BOARD-POSITION#2 BOARD-POSITION#1 QUEEN#2 QUEEN#1
                                 PIECE#2 PIECE#1)
                      (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                      (IF (PIECE-ON-BOARD PIECE#2 BOARD-POSITION)
                         THEN
                         (CONSTRAINT-VIOLATION
                                TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE)
                         ELSE NIL)
                      (IF (PIECE-ON-BOARD PIECE#1 BOARD-POSITION)
                         THEN
                         (CONSTRAINT-VIOLATION
                                TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE)
                         ELSE NIL)
                      (IF (AND (PIECE-ON-BOARD QUEEN#2 BOARD-POSITION#2)
                               (QUEEN-CAPTURE BOARD-POSITION
                                          BOARD-POSITION#2))
                         THEN
                         (CONSTRAINT-VIOLATION
                                 QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN)
                         ELSE NIL)
                      (IF (AND (PIECE-ON-BOARD QUEEN#1 BOARD-POSITION#1)
                               (QUEEN-CAPTURE BOARD-POSITION#1
                                          BOARD-POSITION))
                         THEN
                         (CONSTRAINT-VIOLATION
                                 QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN)
                         ELSE NIL)
```

CHANGES OUT OF STATE-2

```
(LAMBDA (QUEEN-SET)
      (LOCAL (BOARD-POSITION)
            (FOR QUEEN IN-SET QUEEN-SET DO
                (DETERMINE BOARD-POSITION FROM
                         (CHESS-BOARD BOARD-POSITION))
                (LOCAL (BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
                                    PIECE*2 PIECE*1)
                      (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                      (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                         THEN
                         (CONSTRAINT-VIOLATION
                               TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE)
                         ELSE NIL)
                      (IF (PIECE-ON-BOARD PIECE#1 BOARD-POSITION)
                         THEN
                         (CONSTRAINT-VIOLATION
                               TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE)
                         ELSE NIL)
                      (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                             (QUEEN-CAPTURE BOARD-POSITION
                                         BOARD-POSITION*2))
                         THEN
                         (CONSTRAINT-VIOLATION
                                QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN)
                         ELSE NIL)
                      (IF (AND (PIECE-ON-BOARD QUEEN#1 BOARD-POSITION#1)
                             (QUEEN-CAPTURE BOARD-POSITION#1
                                         BOARD-POSITION))
                         THEN
                         (CONSTRAINT-VIOLATION
                                QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN)
                         ELSE NIL)
```

CHANGES INTO STATE-3

```
(LAMBDA (QUEEN-SET)
        (LOCAL (BOARD-POSITION)
               (FOR QUEEN IN-SET QUEEN-SET DO
                    (DETERMINE BOARD-POSITION FROM
                               (CHESS-BOARD BOARD-POSITION))
                    (LOCAL (BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
                                         PIECE*2 PIECE*1)
                           (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                           (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                               THEN
                               (CONSTRAINT-VIOLATION
                                      TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE)
                               ELSE NIL)
                           (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                                    (QUEEN-CAPTURE BOARD-POSITION
                                                   BOARD-POSITION*2))
                               THEN
                               (CONSTRAINT-VIOLATION
                                       QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN)
                               ELSE NIL)
```

CHANGES OUT OF STATE-3

```
(LAMBDA (QUEEN-SET)
        (LOCAL (BOARD-POSITION)
               (FOR QUEEN IN-SET QUEEN-SET DO
                    (DETERMINE BOARD-POSITION FROM
                              (CHESS-BOARD BOARD-POSITION))
                    (LOCAL (BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
                                           PIECE*2 PIECE*1)
                           (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                           (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                               THEN
                               (CONSTRAINT-VIOLATION
                                      TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE)
                               ELSE NIL)
                           (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                                    (QUEEN-CAPTURE BOARD-POSITION
                                                   BOARD-POSITION*2))
                               THEN
                               (CONSTRAINT-VIOLATION
                                       QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN)
                               ELSE NIL]
```

CHANGES INTO STATE-4-1

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                  THEN
                                  (RETURN)))
            (REMOVE QUEEN FROM QUEEN-SET)
            (DETERMINE BOARD-POSITION FROM (CHESS-BOARD BOARD-POSITION))
            (LOCAL (BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
                          PIECE*2 PIECE*1)
                  (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                  (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                     THEN
                     (CONSTRAINT-VIOLATION
                           TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE)
                     ELSE NIL)
                  (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                          (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                     THEN
                     (CONSTRAINT-VIOLATION
                           QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN)
                     ELSE NIL))
            (RECURSIVE-CALL: (QUEENS QUEEN-SET)
```

CHANGES OUT OF STATE-4-1

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION)
           (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                   THEN
                                   (RETURN)))
           (REMOVE QUEEN FROM QUEEN-SET)
           (DETERMINE BOARD-POSITION FROM (CHESS-BOARD BOARD-POSITION))
           (LOCAL (BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
                                   PIECE*2 PIECE*1)
                (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                   THEN
                   (CONSTRAINT-VIOLATION
                              TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE)
                   ELSE NIL)
                (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                         (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                   THEN
                   (CONSTRAINT-VIOLATION
                               QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN)
                   ELSE NIL))
           (RECURSIVE-CALL: (QUEENS QUEEN-SET)
```

CHANGES INTO STATE-4-2-1

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION)
           (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                   THEN
                                   (RETURN)))
           (REMOVE* QUEEN FROM QUEEN-SET)
           (DETERMINE BOARD-POSITION FROM (CHESS-BOARD BOARD-POSITION))
           (LOCAL (BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
                                    PIECE*2 PIECE*1)
                (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                   THEN
                   (CONSTRAINT-VIOLATION
                             TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE)
                   ELSE NIL)
                (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                         (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                   THEN
                   (CONSTRAINT-VIOLATION
                              QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN)
                   ELSE NIL))
           (RECURSIVE-CALL: (QUEENS QUEEN-SET)
```

CHANGES OUT OF STATE-4-2-1

```
(LAMBDA (QUEEN-SET)
        (LOCAL (QUEEN BOARD-POSITION)
               (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                      THEN
                                      (RETURN))
               (REMOVE* QUEEN FROM QUEEN-SET)
               (DETERMINE BOARD-POSITION FROM (CHESS-BOARD BOARD-POSITION))
               (LOCAL (BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
                                        PIECE*2 PIECE*1)
                     (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                     (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                         THEN
                         (CONSTRAINT-VIOLATION
                                    TWO-PIECES-CAN'T-OCCUPY-SAME-SQUARE)
                         ELSE
                         NIL)
                     (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                              (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                         THEN
                         (CONSTRAINT-VIOLATION
                                      QUEEN-CAN'T-CAPTURE-ANOTHER-QUEEN)
                         ELSE
                         NIL))
               (RECURSIVE-CALL: (QUEENS QUEEN-SET)
```

CHANGES INTO STATE-4

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                   THEN
                                   (EXIT SUCCESSFUL))
            (REMOVE* QUEEN FROM QUEEN-SET)
            (FOR ALL (CHESS-BOARD BOARD-POSITION)
                THEREIS
                (LOCAL (BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
                                     PIECE*2 PIECE*1)
                      (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                      (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                         THEN
                         (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                BOARD-POSITION)))
                         (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                         ELSE
                         NIL)
                      (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                              (QUEEN-CAPTURE BOARD-POSITION
                                             BOARD-POSITION*2))
                         THEN
                         (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                BOARD-POSITION)))
                         (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                         ELSE
                         NIL))
                (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET))
                   THEN
                   (LOOP-RETURN CHOICE-ACCEPTED)
                   ELSE
                   (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                          BOARD-POSITION)))
                   (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                THEN
                (EXIT SUCCESSFUL)
                ELSE
                (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                (EXIT UNSUCCESSFUL)
```

CHANGES OUT OF STATE-4

```
(LAMBDA (QUEEN-SET)
    (LOCAL (QUEEN BOARD-POSITION)
        (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                               THEN
                               (EXIT SUCCESSFUL)))
        (REMOVE* QUEEN FROM QUEEN-SET)
        (FOR ALL (CHESS-BOARD BOARD-POSITION)
           THEREIS
           (LOCAL (BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
                                 PIECE*2 PIECE*1)
               (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
               (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                  THEN
                  (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                          BOARD-POSITION)))
                  (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                  ELSE NIL)
               (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                        (QUEEN-CAPTURE BOARD-POSITION
                                 BOARD-POSITION*2))
                  THEN
                  (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                          BOARD-POSITION)))
                  (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                  ELSE NIL))
           (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
              THEN
              (LOOP-RETURN CHOICE-ACCEPTED)
              ELSE
              (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                          BOARD-POSITION)))
              (LOOP-RETURN FORCE-ANOTHER-CHOICE))
           THEN
           (EXIT SUCCESSFUL)
           ELSE
           (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
           (EXIT UNSUCCESSFUL)
```

CHANGES INTO STATE-5-1-1

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                   THEN
                                   (EXIT SUCCESSFUL)))
            (REMOVE* QUEEN FROM QUEEN-SET)
            (LOCAL (BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
                                    PIECE*2 PIECE*1)
                  (FOR ALL (CHESS-BOARD BOARD-POSITION)
                      THEREIS
                      (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                      (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                         THEN
                         (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                   BOARD-POSITION)))
                         (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                         ELSE NIL)
                      (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                               (QUEEN-CAPTURE BOARD-POSITION
                                          BOARD-POSITION*2))
                         THEN
                         (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                   BOARD-POSITION)))
                         (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                         ELSE NIL)
                      (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET))
                                )
                         THEN
                         (LOOP-RETURN CHOICE-ACCEPTED)
                         ELSE
                         (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                   BOARD-POSITION)))
                         (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                      THEN
                      (EXIT SUCCESSFUL)
                      ELSE
                      (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                      (EXIT UNSUCCESSFUL)
```

CHANGES OUT OF STATE-5-1-1

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                   THEN
                                   (EXIT SUCCESSFUL)))
            (REMOVE* QUEEN FROM QUEEN-SET)
            (LOCAL (BOARD-POSITION#2 BOARD-POSITION#1 QUEEN#2 QUEEN#1
                                   PIECE#2 PIECE#1)
                  (FOR ALL (CHESS-BOARD BOARD-POSITION)
                      THEREIS
                      (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                      (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                         THEN
                         (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                    BOARD-POSITION)))
                         (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                         ELSE NIL)
                      (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                               (QUEEN-CAPTURE BOARD-POSITION
                                              BOARD-POSITION*2))
                         THEN
                         (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                    BOARD-POSITION)))
                         (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                         ELSE NIL)
                      (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET))
                                   )
                         THEN
                         (LOOP-RETURN CHOICE-ACCEPTED)
                         ELSE
                         (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                    BOARD-POSITION)))
                         (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                      THEN
                      (EXIT SUCCESSFUL)
                      ELSE
                      (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                      (EXIT UNSUCCESSFUL)
```

CHANGES INTO STATE-5-1

```
(LAMBDA (QUEEN-SET)
    (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION#2 BOARD-POSITION#1 QUE
                QUEEN#1 PIECE#2 PIECE#1)
        (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                THEN
                                (EXIT SUCCESSFUL)))
        (REMOVE* QUEEN FROM QUEEN-SET)
        (FOR ALL (CHESS-BOARD BOARD-POSITION)
            THEREIS
            (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
            (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
               THEN
               (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                    BOARD-POSITION)))
               (LOOP-RETURN FORCE-ANOTHER-CHOICE)
               ELSE NIL)
            (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                     (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
               THEN
               (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                    BOARD-POSITION)))
               (LOOP-RETURN FORCE-ANOTHER-CHOICE)
               ELSE NIL)
            (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
               THEN
               (LOOP-RETURN CHOICE-ACCEPTED)
               ELSE
               (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                    BOARD-POSITION)))
               (LOOP-RETURN FORCE-ANOTHER-CHOICE))
            THEN
            (EXIT SUCCESSFUL)
            ELSE
            (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
            (EXIT UNSUCCESSFUL)
```

CHANGES OUT OF STATE-5-1

```
(LAMBDA (QUEEN-SET)
        (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                    QUEEN*1 PIECE*2 PIECE*1)
              (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                     THEN
                                     (EXIT SUCCESSFUL)))
              (REMOVE* QUEEN FROM QUEEN-SET)
              (FOR ALL (CHESS-BOARD BOARD-POSITION)
                   THEREIS
                   (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                   (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                       THEN
                       (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                            BOARD-POSITION)))
                       (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                       ELSE
                       NIL)
                   (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                            (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                       THEN
                       (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                            BOARD-POSITION)))
                       (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                       ELSE NIL)
                   (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                       THEN
                       (LOOP-RETURN CHOICE-ACCEPTED)
                       ELSE
                       (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                            BOARD-POSITION)))
                       (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                   THEN
                   (EXIT SUCCESSFUL)
                   ELSE
                   (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                   (EXIT UNSUCCESSFUL)
```

CHANGES INTO STATE-5-3

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                QUEEN*1 PIECE*2 PIECE*1)
         (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                THEN
                                (EXIT SUCCESSFUL)))
         (REMOVE* QUEEN FROM QUEEN-SET)
         (FOR ALL (CHESS-BOARD BOARD-POSITION)
            THEREIS
            (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
               THEN
               (LOOP-RETURN FORCE-ANOTHER-CHOICE)
               ELSE NIL)
            (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
            (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                     (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
               THEN
               (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                    BOARD-POSITION)))
               (LOOP-RETURN FORCE-ANOTHER-CHOICE)
               ELSE NIL)
            (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
               THEN
               (LOOP-RETURN CHOICE-ACCEPTED)
               ELSE
               (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                    BOARD-POSITION)))
               (LOOP-RETURN FORCE-ANOTHER-CHOICE))
            THEN
            (EXIT SUCCESSFUL)
            ELSE
            (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
            (EXIT UNSUCCESSFUL]
```

CHANGES OUT OF STATE-5-3

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
             QUEEN*1 PIECE*2 PIECE*1)
         (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                THEN
                                (EXIT SUCCESSFUL)))
         (REMOVE* QUEEN FROM QUEEN-SET)
         (FOR ALL (CHESS-BOARD BOARD-POSITION)
           THEREIS
           (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
              THEN
              (LOOP-RETURN FORCE-ANOTHER-CHOICE)
              ELSE
              NIL)
           (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
           (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                    (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
              THEN
              (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                   BOARD-POSITION)))
              (LOOP-RETURN FORCE-ANOTHER-CHOICE)
              ELSE NIL)
           (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
              THEN
              (LOOP-RETURN CHOICE-ACCEPTED)
              ELSE
              (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                   BOARD-POSITION)))
              (LOOP-RETURN FORCE-ANOTHER-CHOICE))
           THEN
           (EXIT SUCCESSFUL)
           ELSE
           (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
           (EXIT UNSUCCESSFUL]
```

CHANGES INTO STATE-5-5

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                 QUEEN*1 PIECE*2 PIECE*1)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                  THEN
                                  (EXIT SUCCESSFUL)))
            (REMOVE* QUEEN FROM QUEEN-SET)
            (FOR ALL (CHESS-BOARD BOARD-POSITION)
                THEREIS
                (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                   THEN
                   (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                        (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                   THEN
                   (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                   BOARD-POSITION)))
                   (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                   ELSE NIL)
                (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                   THEN
                   (LOOP-RETURN CHOICE-ACCEPTED)
                   ELSE
                   (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                   BOARD-POSITION)))
                   (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                THEN
                (EXIT SUCCESSFUL)
                ELSE
                (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                (EXIT UNSUCCESSFUL]
```

CHANGES OUT OF STATE-5-5

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                   QUEEN*1 PIECE*2 PIECE*1)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                    THEN
                                    (EXIT SUCCESSFUL)))
            (REMOVE* QUEEN FROM QUEEN-SET)
            (FOR ALL (CHESS-BOARD BOARD-POSITION)
                 THEREIS
                 (IF (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)
                     THEN
                     (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                 (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                 (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                          (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                     THEN
                     (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                           BOARD-POSITION)))
                     (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                     ELSE NIL)
                 (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                     THEN
                     (LOOP-RETURN CHOICE-ACCEPTED)
                     ELSE
                     (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                           BOARD-POSITION)))
                     (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                 THEN
                 (EXIT SUCCESSFUL)
                 ELSE
                 (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                 (EXIT UNSUCCESSFUL]
```

CHANGES INTO STATE-5

```
(LAMBDA (QUEEN-SET)
        (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                    QUEEN*1 PIECE*2 PIECE*1)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                   THEN
                                   (EXIT SUCCESSFUL)))
            (REMOVE* QUEEN FROM QUEEN-SET)
            (FOR ALL (APAND (CHESS-BOARD BOARD-POSITION)
                            (NOT (PIECE-ON-BOARD PIECE*2 BOARD-POSITION))
                 THEREIS
                 (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                 (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                          (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                     THEN
                     (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                          BOARD-POSITION)))
                     (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                     ELSE NIL)
                 (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                     THEN
                     (LOOP-RETURN CHOICE-ACCEPTED)
                     ELSE
                     (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                          BOARD-POSITION)))
                     (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                 THEN
                 (EXIT SUCCESSFUL)
                 ELSE
                 (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                 (EXIT UNSUCCESSFUL)
```

CHANGES OUT OF STATE-5

```
(LAMBDA (QUEEN-SET)
    (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
              QUEEN*1 PIECE*2 PIECE*1)
        (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                THEN
                                (EXIT SUCCESSFUL)))
        (REMOVE* QUEEN FROM QUEEN-SET)
        (FOR ALL
            (APAND (CHESS-BOARD BOARD-POSITION)
                  (NOT (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)))
            THEREIS
            (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
            (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                    (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
               THEN
               (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                              BOARD-POSITION)))
               (LOOP-RETURN FORCE-ANOTHER-CHOICE)
               ELSE
               NIL)
            (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
               THEN
               (LOOP-RETURN CHOICE-ACCEPTED)
               ELSE
               (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                              BOARD-POSITION)))
               (LOOP-RETURN FORCE-ANOTHER-CHOICE))
            THEN
            (EXIT SUCCESSFUL)
            ELSE
            (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
            (EXIT UNSUCCESSFUL]
```

CHANGES INTO STATE-6-1

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                QUEEN*1 PIECE*2 PIECE*1)
         (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                 THEN
                                 (EXIT SUCCESSFUL)))
         (REMOVE* QUEEN FROM QUEEN-SET)
         (FOR ALL
             (APAND (CHESS-BOARD BOARD-POSITION)
                    (NOT (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)))
             THEREIS
             (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                      (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                THEN
                (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                ELSE NIL)
             (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
             (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                THEN
                (LOOP-RETURN CHOICE-ACCEPTED)
                ELSE
                (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                    BOARD-POSITION)))
                (LOOP-RETURN FORCE-ANOTHER-CHOICE))
             THEN
             (EXIT SUCCESSFUL)
             ELSE
             (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
             (EXIT UNSUCCESSFUL)
```

CHANGES OUT OF STATE-6-1

```
(LAMBDA (QUEEN-SET)
        (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                  QUEEN*1 PIECE*2 PIECE*1)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                     THEN
                                     (EXIT SUCCESSFUL)))
            (REMOVE* QUEEN FROM QUEEN-SET)
            (FOR ALL
               (APAND (CHESS-BOARD BOARD-POSITION)
                   (NOT (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)))
               THEREIS
               (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                      (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                  THEN
                  (LOOP-RETURN FORCE-ANOTHER-CHOICE)
                  ELSE
                  NIL)
               (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
               (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                  THEN
                  (LOOP-RETURN CHOICE-ACCEPTED)
                  ELSE
                  (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                    BOARD-POSITION)))
                  (LOOP-RETURN FORCE-ANOTHER-CHOICE))
               THEN
               (EXIT SUCCESSFUL)
               ELSE
               (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
               (EXIT UNSUCCESSFUL)
```

CHANGES INTO STATE-6-3

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                QUEEN*1 PIECE*2 PIECE*1)
         (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                THEN
                                (EXIT SUCCESSFUL)))
         (REMOVE* QUEEN FROM QUEEN-SET)
         (FOR ALL
            (APAND (CHESS-BOARD BOARD-POSITION)
                 (NOT (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)))
            THEREIS
            (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                   (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
               THEN
               (LOOP-RETURN FORCE-ANOTHER-CHOICE))
            (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
            (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
               THEN
               (LOOP-RETURN CHOICE-ACCEPTED)
               ELSE
               (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                          BOARD-POSITION)))
               (LOOP-RETURN FORCE-ANOTHER-CHOICE))
            THEN
            (EXIT SUCCESSFUL)
            ELSE
            (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
            (EXIT UNSUCCESSFUL))
```

CHANGES OUT OF STATE-6-3

```
(LAMBDA (QUEEN-SET)
        (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                 QUEEN*1 PIECE*2 PIECE*1)
          (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                 THEN
                                 (EXIT SUCCESSFUL)))
          (REMOVE* QUEEN FROM QUEEN-SET)
          (FOR ALL
              (APAND (CHESS-BOARD BOARD-POSITION)
                    (NOT (PIECE-ON-BOARD PIECE*2 BOARD-POSITION)))
              THEREIS
              (IF (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                       (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2))
                 THEN
                 (LOOP-RETURN FORCE-ANOTHER-CHOICE))
              (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
              (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                 THEN
                 (LOOP-RETURN CHOICE-ACCEPTED)
                 ELSE
                 (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                      BOARD-POSITION)))
                 (LOOP-RETURN FORCE-ANOTHER-CHOICE))
              THEN
              (EXIT SUCCESSFUL)
              ELSE
              (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
              (EXIT UNSUCCESSFUL]
```

CHANGES INTO STATE-6

```
(LAMBDA (QUEEN-SET)
    (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
            QUEEN*1 PIECE*2 PIECE*1)
        (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                               THEN
                               (EXIT SUCCESSFUL)))
        (REMOVE* QUEEN FROM QUEEN-SET)
        (FOR ALL
            (APAND (APAND (CHESS-BOARD BOARD-POSITION)
                          (NOT (PIECE-ON-BOARD PIECE*2 BOARD-POSITION))
                          )
                (NOT (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                          (QUEEN-CAPTURE BOARD-POSITION
                                         BOARD-POSITION*2)
            THEREIS
            (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
            (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                THEN
                (LOOP-RETURN CHOICE-ACCEPTED)
                ELSE
                (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                     BOARD-POSITION)))
                (LOOP-RETURN FORCE-ANOTHER-CHOICE))
            THEN
            (EXIT SUCCESSFUL)
            ELSE
            (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
            (EXIT UNSUCCESSFUL)
```

CHANGES OUT OF STATE-6

```
(LAMBDA (QUEEN-SET)
        (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                  QUEEN*1 PIECE*2 PIECE*1)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                   THEN
                                   (EXIT SUCCESSFUL)))
            (REMOVE* QUEEN FROM QUEEN-SET)
            (FOR ALL
                (APAND (APAND (CHESS-BOARD BOARD-POSITION)
                              (NOT (PIECE-ON-BOARD PIECE*2 BOARD-POSITION))
                       (NOT (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                                 (QUEEN-CAPTURE BOARD-POSITION
                                                BOARD-POSITION*2))
                THEREIS
                (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                    THEN
                    (LOOP-RETURN CHOICE-ACCEPTED)
                    ELSE
                    (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                         BOARD-POSITION)))
                    (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                THEN
                (EXIT SUCCESSFUL)
                ELSE
                (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                (EXIT UNSUCCESSFUL)
```

CHANGES INTO STATE-7

```
(LAMBDA (QUEEN-SET)
      (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                  QUEEN*1 PIECE*2 PIECE*1)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                   THEN
                                   (EXIT SUCCESSFUL)))
            (REMOVE* QUEEN FROM QUEEN-SET)
            (FOR ALL
                (APAND (CHESS-BOARD BOARD-POSITION)
                       (NOT (PIECE-ON-BOARD PIECE*2 BOARD-POSITION))
                       (NOT (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                                 (QUEEN-CAPTURE BOARD-POSITION
                                                BOARD-POSITION*2)
                THEREIS
                (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                    THEN
                    (LOOP-RETURN CHOICE-ACCEPTED)
                    ELSE
                    (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                         BOARD-POSITION)))
                    (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                THEN
                (EXIT SUCCESSFUL)
                ELSE
                (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                (EXIT UNSUCCESSFUL]
```

CHANGES OUT OF STATE-7

```
(LAMBDA (QUEEN-SET)
        (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                    QUEEN*1 PIECE*2 PIECE*1)
              (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                     THEN
                                     (EXIT SUCCESSFUL)))
              (REMOVE* QUEEN FROM QUEEN-SET)
              (FOR ALL
                 (APAND (CHESS-BOARD BOARD-POSITION)
                        (NOT (PIECE-ON-BOARD PIECE#2 BOARD-POSITION))
                        (NOT (AND (PIECE-ON-BOARD QUEEN#2 BOARD-POSITION#2)
                                  (QUEEN-CAPTURE BOARD-POSITION
                                                 BOARD-POSITION#2)
                 THEREIS
                 (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                 (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                     THEN
                     (LOOP-RETURN CHOICE-ACCEPTED)
                     ELSE
                     (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                          BOARD-POSITION)))
                     (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                 THEN
                 (EXIT SUCCESSFUL)
                 ELSE
                 (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                 (EXIT UNSUCCESSFUL)
```

CHANGES INTO STATE-8

```
(LAMBDA
 (QUEEN-SET POSSIBLE-BOARD-POSITIONS)
 (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
        PIECE*2 PIECE*1)
  (TERMINATION-TEST: (IF (EMPTY QUEEN-SET) THEN (EXIT SUCCESSFUL)))
  (REMOVE* QUEEN FROM QUEEN-SET)
  (FOR (BOARD-POSITION) IN-SET POSSIBLE-BOARD-POSITIONS THEREIS
   (MAINTENANCE-ACTIONS: (FOR ALL
              (APAND (CHESS-BOARD BOARD-POSITION)
                     (NOT (AND (PIECE-ON-BOARD QUEEN#3 BOARD-POSITION#3)
                               (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION#3)))
                     (NOT (PIECE-ON-BOARD PIECE#2 BOARD-POSITION)))
              LBIND (BOARD-POSITION#3 QUEEN#3)
              DO (DELETE BOARD-POSITION FROM POSSIBLE-BOARD-POSITIONS))
          (FOR ALL
              (APAND (CHESS-BOARD BOARD-POSITION#3)
                     (NOT (PIECE-ON-BOARD PIECE#3 BOARD-POSITION#3))
                     (QUEEN-CAPTURE BOARD-POSITION#3 BOARD-POSITION)
                     (NOT (AND (PIECE-ON-BOARD QUEEN#2 BOARD-POSITION#2)
                               (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION#2)
              LBIND (BOARD-POSITION#3 PIECE#3)
              DO (DELETE BOARD-POSITION#3 FROM POSSIBLE-BOARD-POSITIONS)))
   (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
   (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
    THEN (LOOP-RETURN CHOICE-ACCEPTED)
    ELSE
    (UNDO-ACTIONS:
     (DENY (PIECE-ON-BOARD QUEEN BOARD-POSITION))
     (MAINTENANCE-ACTIONS: (FOR ALL
              (APAND (CHESS-BOARD BOARD-POSITION)
                     (NOT (AND (PIECE-ON-BOARD QUEEN#3 BOARD-POSITION#3)
                               (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION#3)))
                     (NOT (PIECE-ON-BOARD PIECE#2 BOARD-POSITION)))
              LBIND (BOARD-POSITION#3 QUEEN#3)
              DO (ADD BOARD-POSITION TO POSSIBLE-BOARD-POSITIONS))
          (FOR ALL
              (APAND (CHESS-BOARD BOARD-POSITION#3)
                     (NOT (PIECE-ON-BOARD PIECE#3 BOARD-POSITION#3))
                     (QUEEN-CAPTURE BOARD-POSITION#3 BOARD-POSITION)
                     (NOT (AND (PIECE-ON-BOARD QUEEN#2 BOARD-POSITION#2)
                               (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION#2)
              LBIND (BOARD-POSITION#3 PIECE#3)
              DO (ADD BOARD-POSITION#3 TO POSSIBLE-BOARD-POSITIONS)
    (LOOP-RETURN FORCE-ANOTHER-CHOICE))
   THEN (EXIT SUCCESSFUL)
   ELSE (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
        (EXIT UNSUCCESSFUL)
```

CHANGES OUT OF STATE-8

```
(LAMBDA (QUEEN-SET POSSIBLE-BOARD-POSITIONS)
      (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                  QUEEN*1 PIECE*2 PIECE*1)
         (TERMINATION-TEST: (IF (EMPTY QUEEN-SET) THEN (EXIT SUCCESSFUL)))
         (REMOVE* QUEEN FROM QUEEN-SET)
         (FOR (BOARD-POSITION) IN-SET POSSIBLE-BOARD-POSITIONS THEREIS
             (MAINTENANCE-ACTIONS:
              (FOR ALL (APAND (CHESS-BOARD BOARD-POSITION)
                              (NOT (AND (PIECE-ON-BOARD QUEEN#3
                                                        BOARD-POSITION#3)
                                        (QUEEN-CAPTURE BOARD-POSITION
                                                        BOARD-POSITION#3)))
                              (NOT (PIECE-ON-BOARD PIECE#2 BOARD-POSITION)))
                 LBIND (BOARD-POSITION#3 QUEEN#3) DO
                 (DELETE BOARD-POSITION FROM POSSIBLE-BOARD-POSITIONS))
              (FOR ALL (APAND (CHESS-BOARD BOARD-POSITION*3)
                          (NOT (PIECE-ON-BOARD PIECE*3 BOARD-POSITION*3))
                          (QUEEN-CAPTURE BOARD-POSITION*3 BOARD-POSITION)
                          (NOT (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                                    (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2)
                 LBIND (BOARD-POSITION*3 PIECE*3)
                 DO (DELETE BOARD-POSITION*3 FROM POSSIBLE-BOARD-POSITIONS)
             (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
             (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                 THEN (LOOP-RETURN CHOICE-ACCEPTED)
                 ELSE
                 (UNDO-ACTIONS:
                  (DENY (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                  (MAINTENANCE-ACTIONS:
                   (FOR ALL (APAND (CHESS-BOARD BOARD-POSITION)
                                   (NOT (AND (PIECE-ON-BOARD QUEEN#3
                                                        BOARD-POSITION#3)
                                             (QUEEN-CAPTURE BOARD-POSITION
                                                        BOARD-POSITION#3)))
                                   (NOT (PIECE-ON-BOARD PIECE#2
                                                        BOARD-POSITION)))
                      LBIND (BOARD-POSITION#3 QUEEN#3) DO
                      (ADD BOARD-POSITION TO POSSIBLE-BOARD-POSITIONS))
                   (FOR ALL (APAND (CHESS-BOARD BOARD-POSITION*3)
                               (NOT (PIECE-ON-BOARD PIECE*3 BOARD-POSITION*3))
                               (QUEEN-CAPTURE BOARD-POSITION*3 BOARD-POSITION)
                               (NOT (AND (PIECE-ON-BOARD QUEEN*2 BOARD-POSITION*2)
                                         (QUEEN-CAPTURE BOARD-POSITION BOARD-POSITION*2)
                      LBIND (BOARD-POSITION*3 PIECE*3)
                      DO (ADD BOARD-POSITION*3 TO POSSIBLE-BOARD-POSITIONS)
                 (LOOP-RETURN FORCE-ANOTHER-CHOICE))
             THEN (EXIT SUCCESSFUL)
             ELSE (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                  (EXIT UNSUCCESSFUL)
```

CHANGES INTO STATE-9

```
(LAMBDA (QUEEN-SET POSSIBLE-BOARD-POSITIONS)
        (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                    QUEEN*1 PIECE*2 PIECE*1)
               (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                      THEN (EXIT SUCCESSFUL)))
               (REMOVE* QUEEN FROM QUEEN-SET)
               (FOR (BOARD-POSITION) IN-SET POSSIBLE-BOARD-POSITIONS
                    THEREIS
                    (MAINTENANCE-ACTIONS:
                     (FOR ALL
                          (APAND (CHESS-BOARD BOARD-POSITION*3)
                                 (NOT (PIECE-ON-BOARD PIECE*3 BOARD-POSITION*3))
                                 (QUEEN-CAPTURE BOARD-POSITION*3
                                                BOARD-POSITION)
                                 (NOT (AND (PIECE-ON-BOARD QUEEN*2
                                                           BOARD-POSITION*2)
                                           (QUEEN-CAPTURE BOARD-POSITION
                                                          BOARD-POSITION*2)
                          LBIND (BOARD-POSITION*3 PIECE*3)
                          DO (DELETE BOARD-POSITION*3 FROM
                                 POSSIBLE-BOARD-POSITIONS)
                    (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                    (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)))
                        THEN (LOOP-RETURN CHOICE-ACCEPTED)
                        ELSE
                        (UNDO-ACTIONS:
                         (DENY (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                         (MAINTENANCE-ACTIONS:
                          (FOR ALL
                               (APAND (CHESS-BOARD BOARD-POSITION*3)
                                      (NOT (PIECE-ON-BOARD PIECE*3
                                                           BOARD-POSITION*3))
                                      (QUEEN-CAPTURE BOARD-POSITION*3
                                                     BOARD-POSITION)
                                      (NOT (AND (PIECE-ON-BOARD QUEEN*2
                                                           BOARD-POSITION*2)
                                                (QUEEN-CAPTURE BOARD-POSITION
                                                           BOARD-POSITION*2)
                               LBIND (BOARD-POSITION*3 PIECE*3)
                               DO
                               (ADD BOARD-POSITION*3 TO
                                    POSSIBLE-BOARD-POSITIONS)
                          (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                    THEN (EXIT SUCCESSFUL)
                    ELSE
                    (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                    (EXIT UNSUCCESSFUL)
```

CHANGES OUT OF STATE-9

```
(LAMBDA
 (QUEEN-SET POSSIBLE-BOARD-POSITIONS)
 (LOCAL
  (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2 QUEEN*1
       PIECE*2 PIECE*1)
  (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                      THEN (EXIT SUCCESSFUL)))
  (REMOVE* QUEEN FROM QUEEN-SET)
  (FOR (BOARD-POSITION) IN-SET POSSIBLE-BOARD-POSITIONS
   THEREIS
   (MAINTENANCE-ACTIONS: (FOR ALL
                      (APAND (CHESS-BOARD BOARD-POSITION#3)
                           (NOT (PIECE-ON-BOARD PIECE#3
                                          BOARD-POSITION#3))
                           (QUEEN-CAPTURE BOARD-POSITION#3
                                     BOARD-POSITION)
                           (NOT (AND (PIECE-ON-BOARD QUEEN#2
                                          BOARD-POSITION#2)
                                    (QUEEN-CAPTURE BOARD-POSITION
                                          BOARD-POSITION#2)
                      LBIND (BOARD-POSITION#3 PIECE#3)
                      DO
                      (DELETE BOARD-POSITION#3 FROM
                            POSSIBLE-BOARD-POSITIONS)))
   (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
   (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET)
    THEN (LOOP-RETURN CHOICE-ACCEPTED)
    ELSE
    (UNDO-ACTIONS:
     (DENY (PIECE-ON-BOARD QUEEN BOARD-POSITION))
     (MAINTE.NANCE-ACTIONS: (FOR ALL
                       (APAND (CHESS-BOARD BOARD-POSITION#3)
                            (NOT (PIECE-ON-BOARD PIECE#3
                                           BOARD-POSITION#3))
                            (QUEEN-CAPTURE BOARD-POSITION#3
                                      BOARD-POSITION)
                            (NOT (AND (PIECE-ON-BOARD QUEEN#2
                                           BOARD-POSITION#2)
                                     (QUEEN-CAPTURE BOARD-POSITION
                                           BOARD-POSITION#2)
                       LBIND (BOARD-POSITION#3 PIECE#3)
                       DO
                       (ADD BOARD-POSITION#3 TO
                            POSSIBLE-BOARD-POSITIONS)
    (LOOP-RETURN FORCE-ANOTHER-CHOICE))
   THEN (EXIT SUCCESSFUL)
   ELSE
   (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
   (EXIT UNSUCCESSFUL)
```

CHANGES INTO STATE-10

```
(LAMBDA (QUEEN-SET REMAINING-ROWS REMAINING-COLUMNS
                    REMAINING-NE-DIAGONALS REMAINING-SE-DIAGONALS)
      (LOCAL (QUEEN BOARD-POSITION BOARD-POSITION*2 BOARD-POSITION*1 QUEEN*2
                 QUEEN*1 PIECE*2 PIECE*1)
            (TERMINATION-TEST: (IF (EMPTY QUEEN-SET)
                                   THEN (EXIT SUCCESSFUL)))
            (REMOVE* QUEEN FROM QUEEN-SET)
            (FOR ROW IN-SET REMAINING-ROWS THEREIS
                (FOR COLUMN IN-SET REMAINING-COLUMNS
                 WHEN
                 (PROGN (DETERMINE* NE-DIAGONAL FROM
                                   (CORRESPONDING-NE-DAGONAL ROW COLUMN))
                        (DETERMINE* SE-DIAGONAL FROM
                                   (CORRESPONDING-SE-DIAGONAL ROW COLUMN))
                        (APAND (IN-SET? NE-DIAGONAL REMAINING-NE-DIAGONALS)
                               (IN-SET? SE-DIAGONAL REMAINING-SE-DIAGONALS)
                               ))
                 THEREIS
                 (MAINTENANCE-ACTIONS: (REMOVE* ROW FROM REMAINING-ROWS)
                                       (REMOVE* COLUMN FROM
                                             REMAINING-COLUMNS)
                                       (REMOVE* NE-DIAGONAL FROM
                                             REMAINING-NE-DIAGONALS)
                                       (REMOVE* SE-DIAGONAL FROM
                                             REMAINING-SE-DIAGONALS))
                 (DETERMINE* BOARD-POSITION FROM
                        (CORRESPONDING-BOARD-POSITION BOARD-POSITION
                                                ROW COLUMN))
                 (ASSERT (PIECE-ON-BOARD QUEEN BOARD-POSITION))
                 (IF (SUCCESSFUL (RECURSIVE-CALL: (QUEENS QUEEN-SET
                                                 REMAINING-ROWS
                                               REMAINING-COLUMNS
                                           REMAINING-NE-DIAGONALS
                                           REMAINING-SE-DIAGONALS)
                     THEN (LOOP-RETURN CHOICE-ACCEPTED)
                     ELSE (UNDO-ACTIONS: (DENY (PIECE-ON-BOARD QUEEN
                                                 BOARD-POSITION))
                                (MAINTENANCE-ACTIONS: (ADD ROW TO
                                                 REMAINING-ROWS)
                                                   (ADD COLUMN TO
                                               REMAINING-COLUMNS)
                                                   (ADD NE-DIAGONAL TO
                                           REMAINING-NE-DIAGONALS)
                                                   (ADD SE-DIAGONAL TO
                                           REMAINING-SE-DIAGONALS)
                          (LOOP-RETURN FORCE-ANOTHER-CHOICE))
                 THEN (EXIT SUCCESSFUL)
                 ELSE (UNDO-ACTIONS: (ADD QUEEN TO QUEEN-SET))
                     (EXIT UNSUCCESSFUL)
```

REFERENCES

1. Arsac, J., "Syntactic source to source transforms and program manipulation," *Communications of the ACM* 22 (1), January 1979.

2. Balzer, R., and N. Goldman, "Principles of good software specification and their implications for specification languages," In *IEEE Proceedings Specifications of Reliable Software*, pp. 58-67, Boston, Mass., 1979. Also ISI/RR-80-86.

3. Balzer, R., Set maintenance in a dynamic environment (in preparation).

4. Barstow, D., "A knowledge-based system for automatic program construction," In *Proceedings from the 5th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 382-388, Cambridge, Mass., August 1977.

5. Bauer, F. L., "Programming as an evolutionary process," In *Proceedings of the 2nd International Conference on Software Engineering*, pp. 223-234, San Francisco, Ca., October 1976.

6. Boyle, J. M., and M. Metz, "Automating multiple programs realizations," In *Proceedings of the MRI International Symposium XXIV, Computer Software Engineering*, Polytechnic Press, Brooklyn, N.Y., 1977.

7. Burstall, R. M., and J. L. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM* 24 (1), January 1977, 44-67.

8. Cheatham, T., G. Holloway, and J. Townley, *Symbolic Evaluation and the Analysis of Programs*, Harvard University, Center for Research in Computing Technology, Technical Report 19-78, 1978.

9. Chiu, W., Structure comparison, 1979. USC/Information Sciences Institute (draft report).

10. Cohen, J., "Interpretation of non-deterministic algorithms in higher level languages," *Information Processing Letters* 3 (4), March 1975.

11. Donzeau-Gouge, V., et al., "A structure-oriented program editor: A first step towards computer assisted programming," In *Proceedings of International Computing Symposium 1975*, North Holland, 1975.

12. Feather, M., *"ZAP" Program Transformation System Primer and User's Manual*, University of Edinburgh, Dept. of Artificial Intelligence, Technical Report 54, August 1978.

13. Gerhart, S. L., and L. Yelowitz, "Control structure abstractions of the backtracking programming technique," *IEEE Transactions on Software Engineering* SE-2 (4), December 1976, 285-292.

14. Huet, G., and B. Lang, "Proving and applying program transformations expressed with second-order patterns," *Acta Informatica XI*, 1978, 31-55.

15. Paige, R., and J. T. Schwartz, Expression continuity and the formal differentiation of algorithms. Courant Institute of Mathematical Sciences, New York University, Appendix B from NSF proposal reviewed January 16, 1979.

16. Rich, C., and H. E. Shrobe, "Initial report on a LISP programmer's apprentice," *IEEE Transactions on Software Engineering* SE-4 (6), 1978, 454-467.

17. Schwartz, J. T., "Some syntactic suggestions for transformational programming," *SETL Newsletter* 205, Courant Institute, New York, N. Y., 1978.

18. Standish, T. A., et al., "Improving and refining programs by program manipulation," in *Proceedings of the 1976 ACM Annual Conference*, pp. 509-516, October 1976.

19. Wile, D., Type transformations, 1979. USC/Information Sciences Institute (draft report).

DAT
FILM